Name of Scientific area: SOFTWARE ENGINEERING

USE ARDUINO AS A MODBUS SLAVE OVER RTU

Jose Domingos Senra Simoes, A65378@alunos.uminho.pt, Mechanical Eng. Student

Andre Monteiro Fernandes, A65381@alunos.uminho.pt, Mechanical Eng. Student

Eurico Augusto Rodrigues de Seabra, eseabra@dem.uminho.pt, Mechanical Eng. Professor

ABSTRACT

The purpose of this article is to accomplish various appointed objectives. The objectives of this article include: to become acquainted with the Modbus RTU protocol, to implement the Modbus RTU protocol onto an Arduino Uno microcontroller as a slave and to test the Arduino Uno slave using a free Modbus master program. The Arduino Uno slave must be fully configurable where the user can set all the Modbus communication parameters such as slave ID, parity, communication rate, number of stop bits, etc. The user must also be able to assign values or variables to a particular coil or register regardless of the order of the previously used coils or registers. The Arduino slave must also communicate with the master over a RS232 connection.

KEYWORDS

Arduino, Modbus, slave, RTU, RS232, Serial Communication

1 Introduction

The Modbus RTU protocol is a widely used protocol to communicate with devices. Industrial devices like temperature and pressure sensors, generators, printers and many others use the Modbus protocol to send the readings or to read from a central computer or master device. Since the Arduino Uno microcontroller is so versatile and flexible, it would be interesting to try to use it as a slave device which would read sensor data to send over Modbus and also read data over Modbus to activate or deactivate LEDs or actuators. This would be a great way to learn about the Modbus RTU communication protocol as well as to test its functionality and reliability.

2 Contextualization

2.1 Modbus

Modbus is an industrial protocol that was developed in 1979 by Modicon, Incorporated, to make communication possible between industrial automation systems and Modicon programmable controllers. (Cyburt, n.d.) Originally implemented as an application-level protocol intended to transfer data over a serial layer, Modbus has expanded to include implementations over serial, TCP/IP, and the user datagram protocol (UDP). (National Instruments, 2014) Thus making it a standard communications protocol for electronic devices. (McCrohan, 2011)

The MODBUS protocol follows a client/server (master/slave) architecture where a client requests data from a server. The client can also send data to the server. The client initiates a process by sending a function code that denotes the type of operation to execute. The operation performed by the MODBUS protocol defines the process a controller uses to request access to another device, how it will respond to requests from other devices, and how errors will be detected and reported. The MODBUS protocol establishes a common format for the layout and contents of message fields.

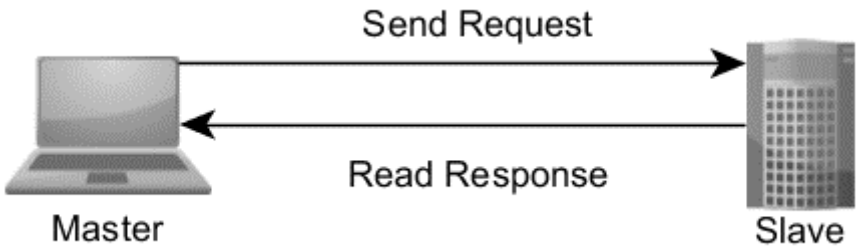

Figure 1 Master and Slave, request and response communication (National Instruments, 2014)

During communication on a MODBUS network, the protocol determines how each controller will know its device address, how a device recognizes a message addressed to it, how the device determines the kind of action to be taken, and how the device extracts any data or other information contained in the message. (National Instruments, 2009) Masters can address individual slaves, or it can send a broadcast message to all slaves where every listening slave receives the message. A slaves replies to all queries that are addressed individually to them, but never respond to broadcast queries. (Cyburt, n.d.)

Controllers communicate using a master/slave technique where only the master can start a query or transactions. The other devices, the slaves, respond to the master by supplying the demanded data or by performing the action demanded in the query. Typically, the master is a human machine interface (HMI) or Supervisory Control and Data Acquisition (SCADA) system (National Instruments, 2014) or other devices which may include host computers(running appropriate application software)/processors and programming panels. Typical slaves include programmable logic controller (PLC), or programmable automation controller (PAC) (National Instruments, 2014), I/O transducer, valve, network drive, or other measuring devices. (Cyburt, n.d.) (National Instruments, 2009)

A query from a master device consists of a slave address (or broadcast address), a function code representing the desired action, any required data pertaining to the function and an error checking field to check for message corruption. (Cyburt, n.d.) When the Master sends a message to the Slave, it is the function code field which informs the server of what type of action to perform. (Real Time Automation, n.d.)

A slave's response is comprised of fields affirming the slave's address, the function executed, any data needed to be returned to the master in accordance with the function and an error checking field for message corruption. In is important to note that the query and response messages are both composed of the slave's address, a function code, any applicable data in accordance with the function code and an error checking field. If no error is detected, then the slave's reply contains the requested data. Otherwise, if an error occurs in the received query, or if the slave is not able to execute the requested action, the slave will reply an exception message as its response. The error checking field of the message frame sanctions the master to affirm that the contents of the received message is valid and not fraudulent. Additionally, parity checking is also applied to each transmitted character in its data frame to check for correctness. (Cyburt, n.d.)

2.1.1 Modbus Data Model

The Modbus-accessible data is kept, in general, in one of four data banks or address ranges in the slave device. These four data banks are holding registers, input registers discrete inputs and coils. Similarly to most of the specification, the terms may change depending on the industry or application at hand. For instance, a coils may be mentioned as a digital or discrete output and a holding register could be denoted to as a output register. The data banks describe the type and access rights of the data contained within. Slave devices have uninterrupted access to the data, which is held locally on the slave devices. The accessible data is usually a subsection of the slave device's main memory. On the other hand, Modbus masters must request access to this data using numerous function codes. The behavior of each block is described in Table 1. (National Instruments, 2014)

Table 1 Modbus Data Model Blocks

Memory Block	**Data Type**	**Master Access**	**Slave Access**
Coils	Boolean	Read/Write	Read/Write
Discrete Inputs	Boolean	Read-only	Read/Write
Holding Registers	Unsigned Word	Read/Write	Read/Write
Input Registers	Unsigned Word	Read-only	Read/Write

The blocks have the ability to permit or restrict access to the diverse data elements and also to deliver simplified means at the application layer to access the different data types. These blocks are totally theoretical. They can occur as separate memory addresses in a specific system, but may also overlap each other. For instance, the first bit of the word represented by holding register one may exist in the same location in memory as coil one. The addressing structure is completely defined by the slave device, its interpretation of each memory block is a significant part of the device's data model. (National Instruments, 2014)

2.1.2 Data Model Addressing

The Modbus protocol describes each block as encompassing "an address space of as many as 65,536 (2^{16}) elements." Modbus describes the address of each data element ranging from 0 to 65,535. Conversely, each data element is numbered starting at 1 to 65,536. That is to say, holding register 1 is in the holding register block at address 0 and coil 308 is at address 307 in the coil memory block, and so on for all addresses. (National Instruments, 2014)

The full memory block ranges allowed by the protocol do not have mandatory usage on any given device. For instance, a slave device may not need to implement coils, discrete inputs, or input registers and in its place only use holding registers 150 to 175 and 200 to 225. This is a perfectly acceptable situation, invalid access attempts would otherwise be treated through exceptions codes. (National Instruments, 2014)

Table 2 Modbus Register Map (Cyburt, n.d.)

Reference	Description
0xxxx	Read/Write Discrete Outputs or Coils. A 0x reference address is used to drive output data to a digital output channel.
1xxxx	Read Discrete Inputs. The ON/OFF status of a 1x reference address is controlled by the corresponding digital input channel.
3xxxx	Read Input Registers. A 3x reference register contains a 16-bit number received from an external source—e.g. an analog signal.
4xxxx	Read/Write Output or Holding Registers. A 4x register is used to store 16-bits of numerical data (binary or decimal), or to send the data from the CPU to an output channel.

The "x" following the leading or prefix character represents a four-digit address located in the user data memory. The leading or prefix character is usually implied by the function code sent in the message and is therefore omitted from the address specifier for a given function. The leading prefix character also identifies the I/O data type implied by the function. (Cyburt, n.d.)

2.1.3 Serial Transmission Modes

The two serial transmission modes accepted by the Modbus protocol are ASCII and RTU Mode. When controllers are structured to communicate on a Modbus network using ASCII (American Standard Code for Information Interchange) mode, every 8–bit byte in a message is sent as two ASCII characters. The main benefit of this mode is that it permits time intervals of up to one second to take place between characters without triggering any error codes. (Modicon, 1996)

When controllers are programmed to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, every 8–bit byte in a message holds two 4–bit hexadecimal characters. The key benefit of this mode is that its greater character density allows for better data throughput than in ASCII mode for the same baud rate. Although, every message must be transferred in an uninterrupted stream of data. (Modicon, 1996)

Table 3 Serial transmission modes ASCII vs RTU (Real Time Automation, n.d.)

	Modbus/ASCII		Modbus/RTU	
Characters	ASCII 0…9 and A..F		Binary 0…255	
Error check	LRC Longitudinal Redundancy Check		CRC Cyclic Redundancy Check	
Frame start	character ':'		3.5 chars silence	
Frame end	characters CR/LF		3.5 chars silence	
Gaps in message	1 sec		1.5 times char length	
Start bit	1		1	
Data bits	7		8	
Parity	even/odd	none	even/odd	none
Stop bits	1	2	1	2

In RTU mode, messages start with a silent interval of at least 3.5 character times. This is easily implemented as a function of the communication baud rate that is used on the network. The first field sent in the packet is the device address. The packet must have all fields transmitted in hexadecimal, 0–9 and A–F. Network devices supervise the network bus uninterruptedly, which includes during the 'silent' breaks. Following the last transmitted character of a message, a similar break of at least 3.5 character times marks the conclusion of the message. A new message can then initiate after this wait time. The whole message frame must be transferred as an continuous stream of data. If a quiet break of more than 1.5 character times arises before the frame transmission ends, the receiving device flushes the incomplete message received and assumes that the following byte received will be the address field of a new message. Likewise, if a new message initiates prior to the termination of the 3.5 character times following the previous message, the receiving device will consider it as a continuation of the preceding message. This event will originate an error, as the value in the final CRC field will not be effective for the concatenated messages. (Modicon, 1996)

A typical message frame is shown below.

START	ADDRESS	Function	DATA	CRC CHECK	END
3.5 char times	8 BITS	8 BITS	N x 8 BITS	16 BITS	3.5 char times

In the Modbus protocol, it is directly stated that "The quantity of registers to be read, combined with all the other fields in the expected response, must not exceed the allowable length of a Modbus messages of 256 bytes." (Courvelle, 2002)

2.1.4 Address Field Handling

The address field of a message frame contains two characters (ASCII) or eight bits (RTU). Valid slave device addresses are between 0 – 247 (decimal). The individual slave devices are assigned addresses between 1 and 247. A master addresses a slave by placing the slave address in the address field of the message. When the slave replies to the query, it places its own address in this address field of the response to let the master know which slave is responding. Address 0 is used as a broadcast address, which all slave devices recognize but do not respond to. (Cyburt, n.d.)

2.1.5 Function Field Handling

The function code is the second field of a message frame; it contains two characters (ASCII) or eight bits (RTU) of data. Valid function codes range between 1 and 255 (decimal). Of these, some codes are applicable to all Modicon controllers, while some codes apply only to certain models, and others are reserved for future use. When a message is sent from a master to a slave device the function code field tells the slave what kind of action to perform. (Cyburt, n.d.) The most used function codes and the most relevant for this project are listed in Table 4.

When the slave responds to the master, it uses the function code field to indicate either a normal response or that some kind of error occurred called an exception response. For a normal response, the slave simply echoes the original function code sent by the master. For an exception response, the slave returns a code that is equivalent to the original function code with its most–significant bit set to a logic 1. In addition to its modification of the function code for an exception response, the slave places a unique code into the data field of the response message. This tells the master what kind of error occurred, or the reason for the exception. The master device's application program has the responsibility of handling exception responses. (Cyburt, n.d.)

Table 4 Function codes and description (Cyburt, n.d.)

01 Read Coil Status	Reads the ON/OFF status of discrete outputs (0X references, coils) in the slave. Broadcast is not supported.
02 Read Input Status	Reads the ON/OFF status of discrete inputs (1X references) in the slave. Broadcast is not supported.
03 Read Holding Registers	Reads the binary contents of holding registers (4X references) in the slave. Broadcast is not supported
04 Read Input Registers	Reads the binary contents of input registers (3X references) in the slave. Broadcast is not supported
05 Force Single Coil	Forces a single coil (0X reference) to either ON or OFF. When broadcast, the function forces the same coil reference in all attached slaves
06 Preset Single Register	Presets a value into a single holding register (4X reference). When broadcast, the function presets the same register reference in all attached slaves.
15 Force Multiple Coils	Forces each coil (0X reference) in a sequence of coils to either ON or OFF. When broadcast, the function forces the same coil references in all attached slaves
16 Preset Multiple Registers	Presets values into a sequence of holding registers (4X references). When broadcast, the function presets the same register references in all attached slaves

2.1.6 Data Field is Handling

All data addresses in Modbus messages are referenced to zero. The first available data item is addressed as item zero. For example, a coil known as coil 0001 in a programmable controller, is addressed as coil 0000 in the data address field of a Modbus message. (Cyburt, n.d.)

The data field is constructed using sets of two hexadecimal digits (ASCII) or using data bytes (RTU) according to the network's serial transmission mode. The data field of the message sent from a master to slave devices holds additional information that the slave must use to be able to carry out the assigned function code. This information can consist of items like coil and register addresses, the quantity of items to be handled, and the count of all the data bytes in the field. For example, if the master requests a slave to read a group of holding registers (function code 03), the data field specifies the starting register and how many registers are to be read. (Cyburt, n.d.)

2.1.7 CRC and Parity Field Handling

The Modbus protocol utilizes two techniques for error checking: parity checking of the data character frame (even, odd, or no parity), and frame checking within the message frame (Cyclical Redundancy Check in RTU Mode, or Longitudinal Redundancy Check in ASCII Mode). (Cyburt, n.d.)

For parity checking, a Modbus slave can be configured for even parity, odd parity or for no parity checking. This setting regulates how the parity bit of the data frame is established. If even or odd parity checking is chosen, the number of "1" bits in the data portion of each character frame is counted. Each character in RTU mode contains 8 bits. The parity bit will then be written as a 0 or a 1, to result in an even (even parity), or odd (odd parity) total number of "1" bits in each message. (Cyburt, n.d.)

The CRC error checking field of a message frame is composed of a 16-bit value (two 8-bit bytes) that contain the product of a Cyclical Redundancy Check (CRC) calculation which was performed on the message contents. The CRC value is computed by the transmitting device and is concatenated to the message as the last item, the low order byte is appended first, followed by the high-order byte. Therefore, the CRC high-order byte is the last byte to be transmitted from the message. The receiving device computes a new CRC value during the reception of the message and tries to match the calculated value to the one received in the CRC field of the message. If the two values differ from one another, an error will generated to notify about the inaccuracy. (Cyburt, n.d.)

If an error does not occur, the data field of the answer from the slave to the master encompasses the data requested by the master. If an error is present, the data field contains an exception code that the master device or application can use to conclude the next action to be taken. (Cyburt, n.d.) The Modbus exception codes available are listed in Table 5

Table 5 Modbus Exception Codes (Cyburt, n.d.)

Code	Exception	Description
01	Illegal Function	The function code received in the query is not allowed or invalid.
02	Illegal Data Address	The data address received in the query is not an allowable address for the slave or is invalid.
03	Illegal Data Value	A value contained in the query data field is not an allowable value for the slave or is invalid.
04	Slave Device Failure	An unrecoverable error occurred while the slave was attempting to perform the requested action.
05	Acknowledge	The slave has accepted the request and is processing it, but a long duration of time is required to do so. This response is returned to prevent a timeout error from occurring in the master.
06	Slave Device Busy	The slave is engaged in processing a long-duration program command. The master should retransmit the message later when the slave is free.
07	Negative Acknowledge	The slave cannot perform the program function received in the query. This code is returned for an unsuccessful programming request using function code 13 or 14 (codes not supported by this model). The master should request diagnostic information from the slave.
08	Memory Parity Error	The slave attempted to read extended memory, but detected a parity error in memory. The master can retry the request, but service may be required at the slave device.

2.1.8 Electrical and Mechanical Interfaces

The electrical interface is what connects the master to all the slaves. There are two most common ways of doing this which is by using the RS 232 or the RS 485 interface. The RS 232 interface is mainly a peer to peer interface which means that it connects the master to just one slave (MODBUS.ORG, 2012), but it is capable of connecting up to 10 slaves to the master. It uses full duplex communication which means that the Rx and the TX packets flow through separate lines. The slaves should not be places further than 15 m. (50 ft.), otherwise the signal could become too weak to be read by the slaves. The RS 232 interface is also very vulnerable in noisy environments, the signal is considered low between 3-15 v. and high between -3-(-15) v. The RS 232 interface consists of typically 3 wire which are the Rx, TX and the GND wires. (Real Time Automation, n.d.)

The RS 485 interface is a multi-drop connection, meaning that it connects to many devices, in this case up to 32 slaves. It uses half duplex communication, so it can only send or receive information at a time. After the master sends the packet, it must switch the drivers so that it can start listening to the slave's response. The interface has high noise immunity and the connection can span up to 350 m. (1000 ft.). The RS485 interface is a 2 wire where one is the RX/TX line and the other is the GND line. (Real Time Automation, n.d.) (MODBUS.ORG, 2012)

The two most common connectors found on devices for these interfaces are the DB-9 connector and normal pin out connectors for manual wiring connections. Although the RJ45 connector could also be used as a serial connector, it is mainly used as a network connector for Modbus TCP (National Instruments, 2013)

2.2 Arduino

2.2.1 Connections

The microcontroller used for this project was an Arduino UNO. The Uno is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins and 6 of these can be used as PWM outputs, 6 analog inputs, a 16 MHz quartz crystal, a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller.

"The Uno can be programmed with the Arduino Software (IDE). The Uno board can be powered via the USB connection or with an external power supply. The power source is selected automatically. Each of the 14 digital pins on the Uno can be used as an input or output, using pinMode(), digitalWrite() and digitalRead() functions in the Arduino IDE. The pins operate at 5 volts and each pin can provide or receive 20 mA as recommended operating condition and has an internal pull-up resistor (disconnected by default) of 20-50k ohm. A maximum of 40mA is the value that must not be exceeded on any I/O pin to avoid permanent damage to the microcontroller." (Arduino, n.d.)

Some pins on the Arduino have specialized functions digital pins 0 and 1 are used for RX and TX serial communication respectfully. They are used to receive (RX) and transmit (TX) TTL serial data. These pins are connected to the corresponding pins of the ATmega8U2 USB-to-TTL Serial chip. Pins 2 and 3 are external interrupts, these pins can be setup to trigger an interrupt in the code when the value becomes low, high or when it starts to rise or fall or even when the value changes using the attachInterrupt() function. This function comes in very handy when connecting optoelectric sensors to the Arduino to count a motor's RPM. Pins 3, 5, 6, 9, 10, and 11have PWM (Pulse Width Modulation) functionality which provide 8-bit PWM output with the analogWrite() function. (Arduino, n.d.)

The Uno has 6 analog inputs, labeled A0 through A5, each of which provide 10 bits of resolution or 1024 different values. By default, the analog pins measure from ground to 5 volts, though is it possible to change the upper end of their range using the AREF pin and the analogReference() function. The AREF pin allows the user to change the reference voltage for the analog inputs. (Arduino, n.d.)

2.2.2 Communication

"The Uno has a number of facilities for communicating with a computer, another Uno board, or other microcontrollers. The ATmega328 provides UART TTL (5V) serial communication, which is available on digital pins 0 (RX) and 1 (TX). An ATmega16U2 on the board channels this serial communication over USB and appears as a virtual com port to software on the computer. The 16U2 firmware uses the standard USB

COM drivers, and no external driver is needed. However, on Windows, a .inf file is required. The Arduino Software (IDE) includes a serial monitor which allows simple textual data to be sent to and from the board. The RX and TX LEDs on the board will flash when data is being transmitted via the USB-to-serial chip and USB connection to the computer (but not for serial communication on pins 0 and 1).” (Arduino, n.d.)

The Arduino environment can be extended through the use of libraries, just like most programming platforms. Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data. A number of libraries come installed with the IDE, but the user can also download or create their own. (Arduino, n.d.) There many Arduino libraries out there that can help and simplify an Arduino sketch. (electronhacks, 2014)

3 Project Development

For this project the Arduino had to be programmed, the physical connections had to be made to the computer’s serial connection and the Arduino slave had to be tested using a Modbus master program.

3.1 Programming

When programing the Arduino Uno to function as a Modbus RTU slave, various libraries were found that allowed for a faster implementation of the protocol to the Arduino microcontroller and seamless connection to other code being executed by the microcontroller that will furnish the values to the coils and registers. Some of the encountered libraries include:

- Simple Modbus (Bester, 2012)
- Simple Modbus Slave (Andreapiovesan, 2014)
- Arduino Modbus RTU Slave Library (GNU Operating System, 2012)
- Modbusino (Stephane, 2012)
- Modbus Master/Slave for Arduino (Smarmengol, 2016)
- Modbus Library for Arduino (Sarmento, 2015)

The library chosen to use is the Arduino Modbus RTU Slave Library. This library was chosen due to the availability of its very well documented information and example Arduino sketches. (electronhacks, 2014) The other libraries were not ideal for various reasons, of which most importantly, most did not support all the necessary Modbus functions (1,2,3,4,5,6,15 and 16) required by the objectives. Most libraries also assign each variable value in the Arduino sketch a register or coil and the numbering is done automatically in the order that the variables appear in the sketch. With these libraries it would not be possible to skip a coil or register and not assign it a value. The Arduino Modbus RTU library does support all the Modbus function delimited in the objectives and allows the user to manually assign a variable value to any register or coil and allows for registers and coils to be skipped and not used.

When programming with the library in the Arduino IDE, all the coil and register addresses used in the program must be declared at the top of the sketch using the *regBank.add(address)* command follower by the address with the correct data prefix (0,1,3 or 4). After which the user can set the initial coil and register values to whichever desired initial value using the command *regBank.set (address, initial value)*. If the initial values are

not important to the user, then this part can be left blank where the program will automatically assign a zero to any register or coil where the initial value has not been set.

In the Arduino IDE, the registers and coil values can be retrieved using the command *regBank.get(address).* The addresses can also be written to using the same command as the initial value command, regBank.set(address, value). These commands can be used directly with variable in the Arduino code. In fact, the register and coil addresses should be viewed as regular variables where they are present in the slave's memory and can be regularly read and written to.

When implementing the Modbus RTU library protocol in an Arduino sketch the user must make sure that the Arduino cycles through the sketch in less time than the timeout response time from the Modbus master. When the master sends a query to the slave, it expects to receive a response from the slave within a certain timeframe otherwise it assumes that the query was not delivered and conveys an error message. If the Arduino slave is executing the sketch for too long, the slave cannot see that there is a query from the master in the buffer and will not respond in time. For this not to happen, the user is adviser to not use delays or create loops in the code where the execution could get interrupted. For more complex programs, multitasking might be necessary to allow various processes to run at the same time by multiplexing. (Earl, Multitasking the Arduino Part 1, 2015) (Earl, Multitasking the Arduino Part 2, 2015)

3.2 Connections

For the Modbus master (computer) to communicate with the Arduino slave, it will need a communication cable. This communication cable will need to be a USB to RS232 serial cable like the one pictured in Figure 2. This USB to Serial Port Adapter provides a bi-directional bridge between USB bus and RS-232 serial port peripheral device. The user must note that when using this type of cable for the first time a driver must be installed, otherwise data will not flow through it. (BotnRoll, n.d.)

Figure 2 USB to Serial cable (BotnRoll, n.d.)

To connect the Arduino slave to the serial cable, an adapter must exist that transforms the TTL (Transistor-Transistor Logic) signal from the RX and TX pins of the Arduino to an RS232 serial signal with a DB9 female connector. The converter in Figure 3 was the one chosen to be implemented. The converter contains 4 pins to be connected to the Arduino slave device. The VCC and the GND pins should be connected to the +5v and the GND pins on the Arduino slave respectfully. The RX and TX pins should be connected to the RX and TX pins on the Arduino board which are digital pins 0 and 1 respectfully. (BotnRoll, n.d.)

The Arduino Uno only has one set of RX and TX pins. When programming the Arduino from the Arduino IDE the Rx pin has to be removed from the TTL to RS232 serial converter, otherwise the Arduino will cause an error since it would be receiving two signals over the same pin thus conflicting with the board programming. A way around this would be to programmatically assign two other pins to be secondary RX and TX pins used for the Modbus communication (Dewey-Hagborg, n.d.)

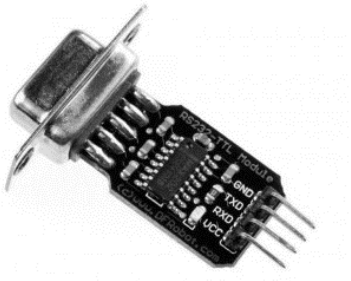

Figure 3 RS232 to TTL converter (and vise-versa) (BotnRoll, n.d.)

Other components that were connected to the Arduino slave device include potentiometers, switches, LEDs and an LCD display. These components were used to represent other components and simulate their signal output (potentiometers and switches) to be interpreted by the Arduino slave. The LCD display showed values programmed into the Arduino sketch and the LEDs would indicate when a potentiometer went over a certain threshold. All these components would be reading and writing to the coil and register addresses. (Arduino, n.d.)

3.3 Testing

To test the Arduino as a Modbus RTU slave, a master device had to be acquired. For this case two very useful Modbus RTU master freeware programs were used. The two programs used are the QModMaster and the Modbus PLC Simulator.

According to Elbar, "QModMaster is a free Qt-based implementation of a Modbus master application. A graphical user interface allows easy communication with Modbus RTU and TCP slaves." QModMaster includes a bus monitor and a bus log for monitoring and examining all traffic flow on the bus. (elbar, 2016)

Modbus PLC Simulator started one weekend as a test program while developing a SCADA/HMI with Modbus RTU and TCP/IP and afterward came in useful testing an embedded gnu-Linux device too. (Zaphodikus, 2009)

Although the Modbus PLC Simulator was a great program to verify the obtained results, it did not have a useful graphical user interface like QModMaster. QModMaster allowed the user to view the bus monitor which shows all the data sent and received over the serial pot. Another major advantage was that it allowed the user to specify what data types the dada was in, i.e. Boolean, integer, etc.

4 Conclusions

Using *QModMaster*, a Modbus master simulator program, communication to and from the Arduino slave was successful. All functions (1,2,3,4,5,15 & 16) were tested and verified on all data types (coils, holding registers, etc.). Initially, the *QModMaster* program indicated that only about 60% of the sent data packets were getting responses. But after working around the delay function in the Arduino IDE, the program cycle time was cut dramatically to below the Modbus master's timeout time which boosted the received number of packets to 100%, making this a very reliable communication protocol.

It is very easy to see why the Modbus protocol is the most frequently used and most implemented protocol in the industry. Because it is so widely used in almost every industrial setting, it has a plethora of useful information available on a public scale. The Modbus protocol might not be the best protocol available today as far as information goes, but it is a very cheap and easily adaptive protocol that can fit any project and uses a minimal amount of wiring to communicate between master and slave.

It was a very gratifying and rewarding experience to be able to communicate to a "homemade" Modbus slave device over the Modbus protocol and have information flow in both directions. This idea could be expanded to control everyday appliances at home, a home security system or even the backyard sprinklers or a greenhouse. For better integration, the user should also create the Modbus master device in a way to allow it to autonomously control all the slaves connected to it.

5 References

Andreapiovesan. (2014, 01 20). *Arduino modbus master/slave communication*. (Automation Corner) Retrieved from https://automationcorner.wordpress.com/2014/01/20/arduino-modbus-masterslave-communication/

Arduino. (n.d.). *Arduino - ArduinoBoardUno*. (Arduino) Retrieved from https://www.arduino.cc/en/main/arduinoBoardUno

Arduino. (n.d.). *Interfacing with Hardware*. (Arduino) Retrieved from http://playground.arduino.cc/Main/InterfacingWithHardware

Arduino. (n.d.). *Libraries*. (Arduino) Retrieved from https://www.arduino.cc/en/Reference/Libraries

Bester, J. (2012, 05 02). *simple-modbus*. (Google Code Archive - Long-term storage for Google Code Project Hosting.) Retrieved from https://code.google.com/archive/p/simple-modbus/

BotnRoll. (n.d.). *RS232-TTL Converter* . (BotnRoll) Retrieved from http://www.botnroll.com/en/rs232-e-rs485/596-rs232-ttl-converter.html?search_query=rs+232&results=9

BotnRoll. (n.d.). *USB To RS232 Serial Port Adapter*. (BotnRoll) Retrieved from http://www.botnroll.com/en/rs232-e-rs485/834-usb-to-rs232-serial-port-adapter.html?search_query=rs+232&results=9

Courvelle, M. (2002, 11 10). *Maximum Amount of Holding Registers per request*. (Control.com) Retrieved from http://control.com/thread/1026161502#1026161502

Cyburt, B. (n.d.). *Introduction to Modbus*. (Acromag, Inc) Retrieved from http://www.automation.com/library/articles-white-papers/fieldbus-serial-bus-io-networks/introduction-to-modbus

Dewey-Hagborg, H. (n.d.). *RS232*. (Arduino) Retrieved from https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232

Earl, B. (2015, 10 13). *Multitasking the Arduino Part 1.* Retrieved from https://cdn-learn.adafruit.com/downloads/pdf/multi-tasking-the-arduino-part-1.pdf

Earl, B. (2015, 01 09). *Multitasking the Arduino Part 2.* Retrieved from https://cdn-learn.adafruit.com/downloads/pdf/multi-tasking-the-arduino-part-2.pdf

elbar. (2016). QModMaster. Sourceforge. Retrieved from https://sourceforge.net/projects/qmodmaster/files/?source=navbar

electronhacks. (2014). Arduino Modbus PLC / RTU. Electron Hacks. Retrieved from http://www.electronhacks.com/2014/04/arduino-modbus-plc-rtu/

GNU Operating System. (2012, 05 05). *arduino-modbus-slave*. (Google Code Archive - Long-term storage for Google Code Project Hosting.) Retrieved from https://code.google.com/archive/p/arduino-modbus-slave/

McCrohan, J. (2011). Arduino Modbus RTU ADC. Dublin, Ireland: dereenigne.org. Retrieved from http://dereenigne.org/arduino/arduino-modbus-rtu-adc

MODBUS.ORG. (2012). Modbus over Serial Line Specification & Implementation guide. MODBUS.ORG.

Modicon. (1996, June). Modbus Protocol Reference Guide. North Andover, Massachusetts. Retrieved from http://modbus.org/docs/PI_MBUS_300.pdf

National Instruments. (2009). Introduction to MODBUS. National Instruments. Retrieved from http://www.micronor.com/products/files/AN112/AN112_NIModbusTutorial.pdf

National Instruments. (2013). Serial Quick Reference Guide. National Instruments. Retrieved from http://www.ni.com/pdf/manuals/371253e.pdf

National Instruments. (2014, 08 01). *The Modbus Protocol In-Depth - National Instruments.* (National Instruments) Retrieved from http://www.ni.com/white-paper/52134/en/

Real Time Automation. (n.d.). *Modbus RTU Protocol Overview*. (Real Time Automation) Retrieved from http://www.rtaautomation.com/technologies/modbus-rtu/

Sarmento, A. (2015, 11 11). *Modbus Library for Arduino*. (GitHub) Retrieved from https://github.com/andresarmento/modbus-arduino

Smarmengol. (2016, 02 27). *Modbus-Master-Slave-for-Arduino*. (GitHub) Retrieved from https://github.com/smarmengol/Modbus-Master-Slave-for-Arduino

Stephane. (2012, 12 11). *Modbusino*. (GitHub) Retrieved from https://github.com/stephane/modbusino

Zaphodikus. (2009). Modbus PLC Simulator. East London, South Africa. Retrieved from http://www.plcsimulator.org/Home